A Language Agnostic Approach to Modeling Requirements: Specification and Verification

Abdelghani Alidra, Antoine Beugnard, Hubert Godfroy, Pierre Kimmel, Gurvan Le Guernic

A Language Agnostic Approach to Modeling Requirements: Specification and Verification

Abdelghani Alidra
Antoine Beugnard
alidrandco@yahoo.fr
antoine.beugnard@imt-atlantique.fr
IMT Atlantique Bretagne-Pays de la Loire
Brest, France

Hubert Godfroy
Pierre Kimmel
hubert.godfroy@capgemini.com
pierre.kimmel@capgemini.com
Capgemini France
Cesson-Sévigné, France

Gurvan Le Guernic
gurvan.le-guernic@def.gouv.fr
gurvan.le_guernic@inria.fr
DGA Maîtrise de l'Information
Inria Rennes – Bretagne Atlantique
Rennes, France

ABSTRACT

Modeling is a complex and error prone activity which can result in ambiguous models containing omissions and inconsistencies. Many works have addressed the problem of checking models' consistency. However, most of these works express consistency requirements for a specific modeling language. On the contrary, we argue that in some contexts those requirements should be expressed independently from the modeling language of the models to be checked. We identify a set of modeling requirements in the context of embedded systems design that are expressed independently from any modeling language concrete syntax. We propose a dedicated semantic domain to support them and give a formal characterization of those requirements that is modeling language agnostic.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; **System modeling languages**; • **Computing methodologies** → **Model verification and validation**.

KEYWORDS

Modeling, domain specific, modeling requirements, validation, formal definition, system decomposition

ACM Reference Format:
Abdelghani Alidra, Antoine Beugnard, Hubert Godfroy, Pierre Kimmel, and Gurvan Le Guernic. 2020. A Language Agnostic Approach to Modeling Requirements: Specification and Verification . In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3419224

1 INTRODUCTION

Models can play a pivotal role in the development process of embedded systems in a contractual relationship. Models can be leveraged during many development phases, in particular during evaluation procedures: in the design phase; or in the validation phase, or even in a certification phase. However, experience shows that modeling is hard and non trivial. Models represent systems at different abstraction levels and from different points of view. They are intended for various stakeholders and for many different tasks. It is therefore important in a contractual relationship that modeling (completeness) requirements clearly elicit what information needs to be part of the models. Then, during the description of large systems, many modeling elements are defined. Among those elements, many relations, dependencies and constraints may exist. Consequently, errors, omissions and inconsistencies can easily arise. In order to be exploitable, models need to avoid such defects. It is then critical to track and correct those errors. Which in turn requires to elicit those constraints and correctness properties, potentially as modeling (soundness) requirements.

Many existing works have addressed this challenge in the literature, mostly the soundness verification part and less the completeness elicitation part. Focusing on UML models, early works attempted to check consistency of class diagrams [5, 41, 58]. These works were extended to handle all or part of OCL constraints. Some works map UML/OCL models to formal specifications expressed in CSP, ObjectZ or B languages [11, 42, 51]. Other approaches express the consistency of UML/OCL models as a satisfiability problem written in relational logic [3, 25, 35, 56], in first order logic [14, 15] or in the HOL/Isabelle language [35]. Some works focused on providing assistance to find conflicts to help modelers debugging inconsistent UML/OCL models [61–63]. Other approaches were interested in checking consistency of models expressed in other modeling languages. For instance, Weckesser et al. address the problem of checking the consistency of Clafer models [59, 60] and Bertram et al. check the consistency of Component and Connectors view models [6, 40]. As a last example, the work introduced by Schrefl and Stumpter [55] presents the particularity of checking behavioral models of the system expressed as behavior diagrams.

A notable characteristic common to the previously cited approaches is that they are closely bound to the language or formalism that is used to model the system under study. However, different companies developing embedded systems may use different modeling languages, such as SysML [44], SDL [32], AADL [20] or even OPM [29] or ARCADIA/Capella [53] to name a few. Hence, modeling requirements expressed by contracting or certification authorities should ideally apply equally to all those modeling languages. Contrary to the previously cited approaches, we aim at exploring a modeling language agnostic approach to express (completeness and soundness) modeling requirements, i.e. our approach defines modeling requirements independently from any concrete

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

modeling language. This allows contracting or certification authorities to impose requirements on delivered models while minimizing the constraints imposed on contractors or vendors regarding the use of their preferred modeling language or development tools and processes.

To cope with this constraint, we follow the agnostic principle advocated for by Le Guernic [37] and introduce a new abstract semantic domain[1] to formally express modeling requirements in an agnostic way. Some parts of this semantic domain definition bears resemblance to IO automata [33, 39]. We also define a set of language agnostic constraints to verify these modeling requirements. In this paper, we focus on requirements related to the functional decomposition of the system. The focus on these particular requirements is motivated by two complementary aspects. Firstly, these requirements reflect the need of procurement agencies to cover some specific verification aspects of the systems, essentially those related to the validation of security equipment. Thus, based on our partners' expertise, these requirements naturally emerged as a prominent and significant subset for this kind of activity. Secondly, as can be noticed from the quick overview presented above, lots of works have been focusing on structural and static models while consistency checking of dynamic and behavioral models has been less considered.

In particular, our current interest is on requirements regarding the behavioral equivalence between: the system; its decomposition; system views; and the environmental solicitations. Indeed, in a contractual or certification relationship, and because of the complexity of the considered systems, it is beneficial if contractors provide (semi) formal models that describe: the global behavior of the system; a first level of behavioral decomposition; and a set of functional views that focus on particularly important or sensitive features of the system. Obviously, the behavioral decomposition and the functional views must be consistent with the global system specification in the sense that the behavior of the system is preserved. Besides, it is important that the models delivered explicit the behavior of the system with regard to environmental solicitations that are relevant for the analysis that has to be made. The verification of those completeness and soundness modeling requirements is a prerequisite to analyze the models further. These verifications are for model analysis what smoke tests [19] are for system testing.

The contributions of this paper are as follows. First, Sect. 2 introduces a specific semantic domain to express language agnostic modeling requirements related to structural and behavioral decomposition, which is exploited in Sect. 3 to formally define, in a modeling language agnostic way, a set of completeness and soundness structural and behavioral decomposition modeling requirements. Related work is discussed in Sect. 4 before concluding.

2 A NEW SEMANTIC DOMAIN FOR ASSESSING MODELS CONSISTENCY REQUIREMENTS

In this section, we introduce a semantic domain for a new family of models we call *Behavioral Decomposition Models* (BDMs). We present an overview of our design choices and of the general structures of BDMs focusing on some particular sub-models. Section 3 focuses on some formal verification we proposed on BDMs.

2.1 Structure

A BDM of a system *sys* in an environment *env* is a formally defined model consisting in several sub-models:

(1) An *Interface Model* (*IM*) listing the interfaces of *sys*, the messages exchanged between *sys* and *env* and the mapping of those messages on the interfaces through events.
(2) An *Environment Model* (*EM*) listing the solicitations by which *env* can challenge *sys* which are taken into account in the current model.
(3) A *System Functions Model* (*SFM*) listing the functions that *sys* can execute (at black box level).
(4) A *System Behaviour Model* (*SBM*) describing how *sys* reacts to the solicitations from *env*.
(5) A set of *Functional View Models* (FVM_i) describing the behavior of *sys* focusing on some messages or functions.
(6) A *System Organs Model* (*SOM*) listing all *sys* organs (ie its first level logical and/or structural components) as well as the structure of these organs: which interfaces they use, which messages they exchange with each other and which functions they can execute.
(7) A set of *Functional Organic Models* (FOM_i) describing the decomposition of black box level functions into calls of organic functions as well as the messages these functions use to communicate.
(8) A set of *Organic Behaviour Models* (OBM_i) describing the behavior of each organ, at a black box level.

We can see that: sub-models 1 to 4 describe the black box behavior of *sys* and its interactions with *env*; with sub-model 5 providing further abstractions of the black box behavior; whereas sub-models 6 to 8 describe the organic decomposition and behavior of *sys* (first level of decomposition).

2.2 Events and solicitations

We use the notion of event to describe interactions between *sys* and *env*, as well as between organs. An *event* is a triple (int, msg, fl) where *int* is an interface, *msg* is a message and fl is a flow direction (either `in` or `out`). This allows the same message (resp. interface) to be used in several interfaces (resp. with several messages). The flow direction allows us to specify interfaces that are both input and output interfaces. We note $\overline{Eve^{sys}}$ the set of events between *sys* and *env*.

Given some sets of events noted E, we define the set of solicitations *Sol* by: $Sol \subseteq \{\ [or(E)|not(E)]^*\ \mid\ E \subseteq \overline{Eve^{sys}}\ \}$.

A solicitation is a sequence of sets of events, each set being in the scope of an operator *or* or an operator *not*. In each set, the events are all tagged with either `in` or `out`. *Solicitations are a completeness requirement* stating for which possible solicitations *BDM* must explicit the behavior of *sys*. The intuitive semantics of such sets is as follows:

- A set of `in`-events in the scope of *or* means that *env* solicits *sys* with one of these events. *BDM* must explicit the behavior

[1] Due to lack of space, the mapping of General Purpose Modeling Languages, such as SysML, to this semantic domain is not described in this paper.

of *sys* for at least one of these events at this point of the solicitation.
- A set of out-events in the scope of *or* means that *env* will wait for at least one of these events from *sys* in order to continue the solicitation. If *sys* produces such output, *BDM* must explicit the behavior of *sys* for the rest of the solicitation.
- A set of out-events in the scope of *not* means that *env* will not continue the solicitation after receiving this answer. As long as *sys* produces no such output and does not react to another event, *BDM* must explicit the behavior of *sys* for the rest of the solicitation.

For instance, the following solicitation: $S = or(e_1^i, e_2^i) \cdot not(e_1^0) \cdot or(e_2^o) \cdot or(e_2^i)$ where the e_n^i are input events and e_n^o are output events, means that *BDM* must explicit the behavior of *sys* from every initially reachable state, for at least a case where: *env* sends either e_1^i or e_2^i; then, from every state reachable without additional environment solicitation where *sys* has replied e_2^o without replying e_1^o before, for at least a case where *env* sends e_2^i.

In some way, the set of solicitations contained in the *BDM* model describes all the scenarii for which we want our model to explicit the behavior of the system.

2.3 Behavior

The behavior of *sys*, described in *BDM* is a set of *executions*, that is to say *transitions* from system state to system state. Each transition is labeled by:

- at most one input event e^i representing the input event that can trigger the transition,
- a set F of functions representing the functions that *sys* executes when going through this transition,
- a set E of output events representing the outputs produced by *sys* when going through this transition.

In the general case, we will note $e/F/E$ the label of such a transition. In the case where no event triggers the transition, we will note $\perp$ instead of e.

Note that this structure does not indicate the order of execution and emission between the functions of F and the messages of E: we just state that such functions and messages are produced, possibly those messages are the result of these functions, but it is not guaranteed by the model. Similarly, we can suppose that the triggering event e^i caused functions and messages to be produced, but it is not guaranteed. This nondeterminism is necessary in order to allow some implementation freedom while designing the system, and perform retro-engineering.

In paragraph 3.1, we will explain how we propose to check that the behavior of *sys* is coherent with *Sol* (meaning it always specifies a behavior for each of those solicitations).

This semantic domain for behaviors bear resemblance to the semantic domain of automata, notably of Input/Output (IO) automata [17, 33, 36, 39] and particularly partial order IO automata [7]. However, for those semantic domains, the set of executions is a secondary entity derived from an automaton, whereas in our case the set of executions is a primary entity in the semantic domain, for which there may not be a corresponding automaton. In particular, the set of executions of an automaton is closed by "concatenation"[2], which is not the case for the set of executions of our semantic domain. This is needed for cases where the reason for an execution to be possible or not is not explicit at the level of abstraction used for *sys*'s behavior specification, or if *sys*'s behavior is syntactically specified by a set of sequence diagrams [43, §17.8 p.595] or message sequence charts [30, 31]. Moreover, the general meaning of an automaton's executions set is that, if an execution does not belong to this set then this execution is not a possible execution of the system represented by the automaton. In our semantic domain, if an execution x does not belong to *sys*'s executions set, then it only means that the fact that *sys* can or can not show behavior x is unspecified. If one wants to specify that "x" is not a possible execution of *sys*, then the error "handling" behavior of *sys* on x must be explicitly specified and "x" needs to be explicitly added to *sys*'s executions set with final states being error states, while x must be "covered" by a solicitation in *Sol* (this completeness aspect is detailed in Sect 3.1). However, in this paper and for conciseness purposes, the syntax used to define this set will often be an automaton.

2.4 Views

Due to the complexity of some systems, it is necessary to have tools to focus on some particular aspects of it. We make use of the notion of *views*. Each FVM_i specifies the behavior of *sys* in the same way that *BDM* did, except that it removes or merges some states and/or transitions according to certain *input messages of interest* and *functions of interest*. The effects and executability of these items of interest have to be emphasized by those *FVM*s.

For now, our work does not provide methods to design such views [3] but it must check that:

- the behavior of *sys* according to the FVM_i does not contradict the behavior described by *BDM*;
- FVM_i contains the exact necessary amount of information to study the items of interest.

To check this, we use 4 properties characterizing relationships between a trace of execution t and a function f:

- **Permission:** t allows f if t ends with f and is minimal (there is not subtrace[4] of it leading to f);
- **Requirement:** t is required for f if it is a subtrace of every trace allowing f and it is maximal (there is no trace having t as a subtrace that verifies this);
- **Inhibition:** t inhibits f if there is a trace t' leading to f and adding[5] t into t' does not lead to f and it is minimal (temporary disabling);
- **Blocking:** t blocks f if there is a trace t' leading to f and adding t into t' cannot be extended[6] into a trace leading to f and it is minimal (permanent disabling).

[2] If the two executions $x_{1a}.x_{1b}$ and $x_{2a}.x_{2b}$ belong to the executions of an automaton A, and x_{1a} and x_{2a} finish in the same state where x_{1b} and x_{2b} start, then $x_{1a}.x_{2b}$ and $x_{2a}.x_{1b}$ are also executions of A

[3] In the context of the dialog between project management and designers, managers would describe the item of interest they want to examine and designers would use their knowledge of the (future) product to produce the view models.

[4] Subtrace is an order relation akin to scattered subwords.

[5] As a scattered subword.

[6] t extends t' if t' is a subtrace of t.

We then use those properties both with traces containing events of interest and functions of interests, and we ensure that the view preserves those properties with regards to the general behavior of the system. We also reach minimality by ensuring that only elements relevant to those properties appear in the view.

We will not give any further details on this method in the scope of this article. The rigorous definition of system views is complex enough to be the topic of another publication on its own.

2.5 Organic decomposition

The decomposition of *sys* is described by several models. We call *organ* each component of *sys* studied by our model[7].

The interfaces, functions and behavior of each organ is described in the same fashion as *sys* was described. The events concerning each organ are described through its interfaces and a set of messages that organs can exchange with each other. Finally, the functions of *sys* are decomposed into sub-functions assigned to different organs.

In order for the whole BDM to be coherent, we must check that the behaviors of all the organs put together are consistant with the behavior of *sys* as described in *BDM*. Moreover, all those behaviors must be coherent with the functions decompositions. In paragraph 3.2, we will explain how we propose to check these constraints.

3 FOCUS ON SOME FORMALIZATION CHALLENGES

3.1 Requirements on system solicitations

In this section, the intuition behind system solicitations and the corresponding requirements is first presented. Then, we gradually introduce the corresponding mathematical formulation based on the notion of coherence.

3.1.1 Checking the conformance of system's model with solicitations. In the process of designing models in a contracting or certification relationship, it is important for the model not only to be structurally valid, but also to describe the system with enough precision that it can be used for analysis. In that regard, contractors must identify precisely the perimeter of the system behavior being modeled. This perimeter is not necessarily complete. It may be desirable in a first design or evaluation stage to only model the nominal behavior of the main features of the system, and not behaviors related to secondary features or error handling. One way to do that is to specify for which stimuli from the system environment the behavior of the system should be described in the corresponding model. In some cases, such as when ignoring behaviors after error messages, those environment stimuli may depend on system's responses.

In this perspective, the notion of solicitation is introduced. A solicitation specifies a sequence of event exchanges between the system and its environment. Input events (noted e^i) specify the stimuli that the environment submits to the system and for which the model must explicitly describe the behavior of the system. Output events (noted e^o) specify the reaction/answer of the system that determines the evolution of the rest of the solicitation.

Two categories of requirements on output events can be distinguished. The first category concerns `out`-events that the system could generate and that are necessary to continue the solicitation. A typical example of such an `out`-event is a message sent by the system to notify that a connection or login has been granted and that (in that case only) the rest of the solicitation can be submitted.

The second category of requirements on `out`-events concerns those events that the system could generate and that prohibit the continuation of its solicitation by the environment. A login error is a typical example of such an event. The corresponding semantics expresses that the solicitation may continue to be submitted to the system if no such events are generated by the system but that no verdict on the conformance of the behavior with the solicitation can be given if such an event is generated.

Formally, we define a solicitation as a sequence of sets of `in`-events and sets of `out`-events. These events are within the scope of two distinct operator that indicate whether they are required to appear (the *Or* operator) or required not to appear (the *Not* operator) in the model of the system.

Intuitively, a system behavior model X^{sys} (a set of executions, see 2.3) conforms to a single solicitation s if and only if, starting from the initial state, X^{sys} fully specifies the behavior of the system after the consumption of any sequence of `in`-events specified in the solicitation whenever the "conditions" on the occurrence (or not occurrence) of `out`-events is met.

To verify such a requirement, we introduce the notion of coherence of a solicitation with a set of executions of the system. Without loss of generality, one can consider that an execution reflects a system's state and can be understood as so in the following definitions.

Informally, coherence of a set of executions X and a solicitation s means that: 1) the direct behavior of the system is specified by its model X when solicited with e^i, the first `in`-event in s; and 2) for any execution $x \in X$ that checks e^i, any prolongation of x by $\perp$-transitions (transitions without trigger) that generates the expected system `out`-events following e^i in s leads into a state where the coherence of the rest of the solicitation can be checked. In this sense, `out`-event can be seen as filtering conditions of system executions that need to be checked.

3.1.2 Solicitations with single `in`*-event sequences.* For simplicity's sake, let's give a first definition of conformance and coherence using solicitations composed of sequences `in`-events within *Or* operators only, with the semantics introduced in section 2.2.

Let's also, for illustration purposes, consider a simple system model consisting in all the paths starting in q_0 in the automaton depicted in figure 1 and the solicitations $s = Or(a)$ and $s' = Or(a).Or(b)$

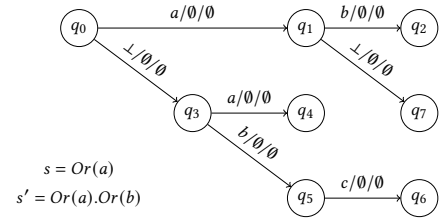

Figure 1: Executions and `in`-event solicitation sequences

[7] This can be either an actual physical component or a logical entity.

Intuitively, one can check that the system's model X^{sys} conforms to the solicitation s because starting from the initial state q_0, X^{sys} specifies the behavior of the system in case it is solicited with a, the first in-event of s. It is worth noting that X^{sys} specifies this behavior including for the case where the system takes the execution path starting with $\perp$ and is in state q_3.

However, it is noticeable that the model X^{sys} does not conform with the solicitation s'. Indeed, this model does not specify the behavior of the system when solicited with the b event after it evolves within the execution path starting with $\perp$ then accepts a (thus being in state q_4). The model does not specify neither the behavior of the system when it is solicited with b after it evolves with the execution path composed of the event a then $\perp$ (corresponding to state q_7).

A first formulation. With this quick example we have outlined the tools we need to formalize conformance between a set of solicitations and a set of executions. Let us consider that we have already examined the beginning of a solicitation containing only in-events, and there remains a solicitation $Or(e^i) \cdot s$[8]. Let us also consider that, by exploring the beginning of the solicitation, we have reached a certain set of states, which correspond to a set of executions X. Then, to check $Or(e^i) \cdot s$ we need to verify two things: 1) we have a transition starting with e^i from any state of X; and 2) s can be checked in any state reachable from the states of X, through transitions bearing e^i, and with any number of $\perp$-transitions afterwards.

We see that in both cases we need to characterize the set of states (that is, of executions) that can be reached from X through transitions bearing e^i. Therefore we introduce: $Next_{elem}(x, e) = \{ x \cdot \delta \in X^{sys} \mid in(\delta) = e \}$

To characterize 2), we also need to define the set of states (i.e. executions) reachable starting with a certain set of states and getting through $\perp$-transitions only. This can be defined this way: $Reach_{in}(X) = \{ x' \cdot \delta \cdot x \in X^{sys} \mid x' \cdot \delta \in X \wedge x = \delta_1 \cdot \ldots \cdot \delta_n \wedge \forall i, in(\delta_i) = \perp \}$

We then can write an inference rule for the coherence between $Or(e^i) \cdot s$ and X this way:

$$\frac{\forall x \in X, Next_{elem}(x, e^i) \neq \emptyset \qquad coh_{in}(s, X')}{coh_{in}(Or(e^i) \cdot s, X)}$$

with $X' = Reach_{in}(\cup_{x \in X} Next_{elem}(x, e^i))$.

This definition is easily extendable to the case where the Or operator contains a set of possible events instead of a single one. In that case we will have $Next(x, E) = \{x \cdot \delta \in X^{sys} | in(\delta) \in E\}$, and then:

$$\frac{\forall x \in X, Next(x, E^i) \neq \emptyset \quad coh_{in}(s, Reach_{in}(\cup_{x \in X} Next(x, E^i)))}{coh_{in}(Or(E^i) \cdot s, X)}$$

We also have the following halting cases, which correspond respectively to an empty solicitation (which is coherent with any execution in the system) and an empty set of execution (which indicates an empty set of paths to check):

$$\textsc{CohSEmpty}\ \frac{X \subseteq X^{sys}}{coh_{in}(\epsilon, X)} \qquad \textsc{CohXEmpty}\ \frac{}{coh_{in}(s, \emptyset)}$$

[8] $s = e \cdot s'$ is freely used to say that s is a solicitation that starts with e and is followed by s'. Similarly, $x \cdot \delta$ denotes the execution starting by x and ending with transition δ. We also use $in(\delta)$ to designate the trigger of δ.

Finally, in order to check that a set S of in-solicitations conforms to the system behavior (which we will note $Conf(S)$), we need to initialize the X set. At the beginning, the states we consider are those reachable from the initial states with $\perp$-transitions, without matching any in-events, that is: $Init_{in} = \{ x \in X^{sys} \mid x = \delta_1 \cdot \ldots \cdot \delta_n \wedge \forall i, in(\delta_i) = \perp \}$

We then have $Conf(S) \Leftrightarrow \forall s \in S, coh_{in}(s, Init_{in})$.

3.1.3 Filtering with output events. The definition of coherence is now extended to take out-events into account. To avoid ambiguous notations, we will note $OrIn$ for Or operators with in-events and $OrOut$ for Or operators with out-events. Please recall that out-events are used to determine in which case a solicitation should continue to be submitted to the system. Then, one can understand out-events in a solicitation as filters of the execution paths (or system states) that need to be considered for the remaining solicitation.

For instance, referring to the model depicted in figure 2, it is easy to verify that this model conforms with the solicitations $s = OrIn(a).OrOut(w, y).OrIn(c, b)$ where a,b and c are in events and y and w are out-events. Indeed, after accepting a, the model specifies the behavior of the system in case the out-event w is generated which corresponds to the case where the system evolves along the execution path $x = q_0 \overset{a/\emptyset/w}{\rightarrow} q_1 \overset{b/\emptyset/\emptyset}{\rightarrow} q_2$. Recall that the model does not need to specify this behavior for the other execution paths (for instance the one starting with a $\perp$ transition) because the system does not generate the out-event y or w required by the solicitation sequence after its acceptance of a.

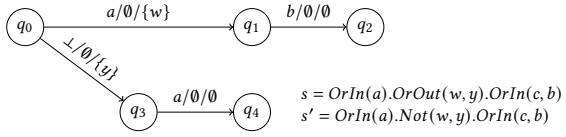

Figure 2: Executions and solicitation sequences

On the contrary, it is also ascertainable that the model of the system does not conform with $s' = OrIn(a).Not(w, y).OrIn(c, b)$ because the model does not specify the behavior of the system in case it is solicited with input c nor with input b after it accepts input a while it generates none of the out-events within the Not operator in s. This behavior corresponds to the execution path $x = q_0 \overset{\perp/\emptyset/y}{\rightarrow} q_3 \overset{a/\emptyset/\emptyset}{\rightarrow} q_4$ (but not to the execution path $q_0 \overset{a/\emptyset/w}{\rightarrow} q_1$ since this path generates the out-event w which is in the scope of a Not operator after interaction a).

With respect to this semantics, we adapt the definition of coherence with regard to the idea of filtering target executions (states) after interaction on an in-event using out-events. First, we see that we need to separate series of in-events and out-events in a solicitation: we will filter the accessible states with all the out-events starting the solicitation and use the Coh relationship with all the in-events following. Thus, we must separate solicitations between all the out-events starting them and the rest, starting by an in-event. For all solicitation s, we define two function inF and $outF$ such that $s = outF(s) \cdot inF(s)$ and $outF(s)$ is the maximal prefix of s beholding only out-events. Therefore, if s starts with

an in-event, $outF(s) = \epsilon$ and $inF(s) = s$. If s contains only out-elements, then $outF(s) = s$ and $inF(s) = \epsilon$. We will then need to use the $outF$ part of the solicitation to filter the traversing of $\perp$-transitions that we performed in the *Reach* sets. For this, we must have a way to check whether an execution matches a sequence of out-events. For sets of out-events $E_1, \ldots, E_n$ and an execution x, we will note $E_1, \ldots, E_n \vDash x$ whenever $x = \delta_1 \cdot \ldots \cdot \delta_m$ and there exists an increasing function $\phi : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that $\forall i \in \{1, \ldots, n\}, out(\delta_{\phi(i)}) \cap E_i \neq \emptyset$. The $\vDash$ relation ensures that the elements of the E_i sets can be found in the output events of x in the right order.

We consider a solicitation composed only of out-events. We then define a relation *check* by the following inferences:

$$\textsc{CheckAx}\ check(x, \epsilon)$$

$$\textsc{CheckOr}\ \frac{E_1, \ldots, E_n \vDash x_1 \quad check(x_2, s)}{check(x_1 \cdot x_2, OrOut(E_1) \cdot \ldots \cdot OrOut(E_n) \cdot s)}$$

We see that this relation checks whether the *OrOut* solicitations appear in some prefix of the execution we consider. We can easily extend this relation for the *Not* solicitations. The simplest case is when the solicitation contains *only* occurrences of *Not*. Then we must check that no prefix of x match those events :

$$\textsc{CheckNot-}\epsilon\ \frac{\forall x_1 \cdot x_2 = x, E_1, \ldots, E_n \nvDash x_1}{check(x, Not(E_1) \cdot \ldots \cdot Not(E_n))}$$

The last case is a bit more complicated. When a series of *Not* is followed by *OrOut*, then events in the scope of the *Not*s must never be seen before an occurrence of the event in *OrOut* and at least one of these occurrences must exist:

$$\textsc{CheckNotOr}\ \frac{\begin{array}{l}\forall x_1 \cdot x_2 = x, \\ \quad check(x_2, OrOut(E_{n+1} \cdot s) \Rightarrow E_1, \ldots, E_n \nvDash x_1 \\ \exists x_1 \cdot x_2 = x, check(x_2, OrOut(E_{n+1} \cdot s)\end{array}}{check(x, Not(E_1) \cdot \ldots \cdot Not(E_n) \cdot OrOut(E_{n+1} \cdot s)}$$

With the *check* relation fully defined, we can extend our previous sets using it. We want to define the set of states we can reach, not only by traversing $\perp$-translations, but also by checking that those translations match a series of out-events. We have then this new definition:

$$Reach(X, s) = \left\{ x' \cdot \delta \cdot x \in X^{sys} \;\middle|\; \begin{array}{l} x' \cdot \delta \in X \\ x = \delta_1 \cdot \ldots \cdot \delta_n \\ \forall i, in(\delta_i) = \perp \\ check(\delta \cdot x, s) \end{array} \right\}$$

We can then extend our definition of coherence:

$$\textsc{CohOr}\ \frac{\forall x \in X, Next(x, E^i) \neq \emptyset \quad coh(inF(s), Reach((\cup_{x \in X} Next(x, E^i)), outF(s)))}{coh(Or(E^i) \cdot s, X)}$$

We also need to update the set of initial states with our checking of out-events prefixes: $Init(s) = \{\ x \in X^{sys} \mid x = \delta_1 \cdot \ldots \cdot \delta_n \ \wedge \ \forall i, in(\delta_i) = \perp \ \wedge \ check(x, s)\ \}$

And lastly we have:

$$Conf(S) \Leftrightarrow \forall s \in S, coh(inF(s), Init(out(s)))$$

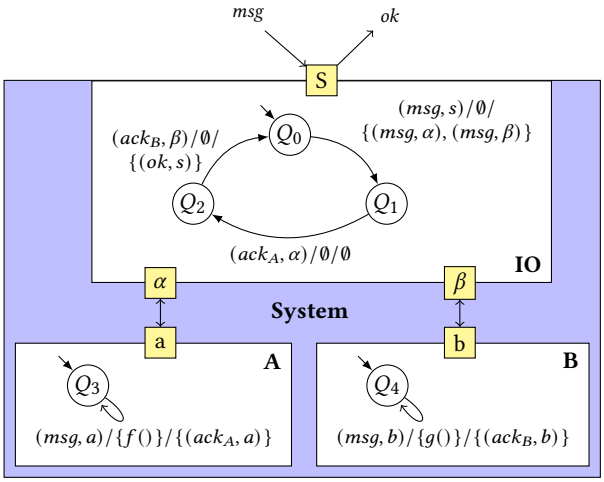

Figure 3: Illustrative example

Finally, we have written a formal relation that checks whether the set of executions X^{sys} which models the system's behavior covers the set of solicitations of interest for the BDM.

3.2 Requirements on organic decomposition

In this section, modeling requirements specifically related to the organic decomposition of the system are introduced. An informal presentation of these requirements is first given. Then, we present the necessary data structures and algorithms to support reasoning on behavioral decomposition.

3.2.1 Approach. Our aim with these requirements is to check that the behavior of all the organs of the models, composed together, is similar to the behavior of the whole system.

The behavior of each organ (as well as the whole system) is described with a set of executions X. Each execution $x \in X$ is a finite sequence of transitions δ_i (the ending state of each being the starting state of the next one). Each transition δ is of the form $\delta = x \xrightarrow{e/F/E} x'$ where x and x' are the source and target states, e is a trigger event (which should be an input event, marked with in), F is a set of functions triggered by e and E is a set of effect events (which should be output events, marked with out). Remember that we do not want to prejudge the order between the elements of both E and F (this may be unknown or intentionally unspecified).

Input/Output Automata [39] are used to address similar problems [16]. However, our sets of executions are strictly more expressive than those extracted from Input/Output Automata (see Sect. 2.3) and their solutions are not directly applicable to our case. Therefore, we introduced a slightly different form of parallel composition in which messages between organs are buffered. This buffer does not need to be local to each organ, because our events are identified with interfaces, thus a given event could only be happening at a specific organ. Consequently, we just use a pool of messages as a buffer. Initially, this pool will be filled with an outside event which triggers the whole organic execution. Then, each transition consists in picking an event in the pool, transitioning any organ

that can transition with this event as a trigger, and then updating the pool with every output events sent by those organs.

3.2.2 Illustrative example. To explain our method, we will use the example presented in Figure 3 which represent both the system's organs and their behavior. The system is composed of three organs, noted IO, A and B. On the black-box level, the system has a single transition behavior of the following form:

$$\delta_{sys} = q \xrightarrow{(msg,s)/\{\phi\}/\{(ok,s)\}} q'$$

Each organ also has a behavior, described in Figure 3. From this description, we can infer the general behavior that is supposed to happen while executing the ϕ function:

(1) The environment sends msg to the system, on the interface s which is actually attached to the organ IO.
(2) IO enters state Q_1 and sends msg on both the α and β interfaces.
(3) msg is received by A on a and by B on b.
(4) Both A and B executes their internal function (respectively f and g) and each send an ack message back to IO.
(5) Once IO has received ack_A, it enters Q_2 and waits for ack_B.
(6) Once IO has received ack_B, it sends out ok and gets back to its initial state.

We will show how our method allows to check that the organic behavior is a correct decomposition relatively to the execution δ_{sys}.

3.2.3 Multi-transitions. We acknowledge that our model gives us the connections between organs. In this context, this means that given an output event e, we can get the corresponding input event $\tilde{e}$ that can receive this message. We note $\tilde{E} = \{\tilde{e} | e \in E\}$. In our example, we have for instance $\widetilde{(msg,\alpha)} = (msg, a)$ because a and α are two interfaces talking to each other.

Moreover we note $Events$ the set of events appearing between two organs of our model. For instance, in our example, $(msg, a) \in Events$ but $(msg, s) \notin Events$ because s is an outer interface.

As we deal with executions instead of automata, we will use prefixes of executions instead of states[9]. We consider a set of n organs. Each organ has a behavior described by a set of executions X_i.

In the case where executions are described by an automaton, states of this automaton can be used instead of prefixes.

Let $\mathbb{Q} = \{x_1, \ldots, x_n\}$ and $\mathbb{Q}' = \{x'_1, \ldots, x'_n\}$ be elements of $pref(X_1) \times \ldots \times pref(X_n)$. We note $\mathbb{Q} \overset{e/F/E}{\Rightarrow} \mathbb{Q}'$ the relation defined by:

$$\mathbb{Q} \overset{e/F/E}{\Rightarrow} \mathbb{Q}' \quad \overset{def}{\Leftrightarrow} \quad \exists i, \begin{cases} x_i \overset{e/F/E}{\Rightarrow} x'_i \\ \forall j \neq i, x_j = x'_j \end{cases}$$

In our example, the prefixes are the set of states. All the possible sets are (Q_0, Q_3, Q_4), (Q_1, Q_3, Q_4) and (Q_2, Q_3, Q_4). According to our previous definition, we have $\mathbb{Q} \overset{e/F/E}{\Rightarrow} \mathbb{Q}'$ if one of the states can change state with the given transition, and the other states do not move. Thus, the following transitions are possible:

$$(Q_0, Q_3, Q_4) \xrightarrow{(msg,a)/\{f()\}/\{(ack_A,a)\}} (Q_0, Q_3, Q_4)$$
$$(Q_0, Q_3, Q_4) \xrightarrow{(msg,s)/\emptyset/\{(msg,\alpha),(msg,\beta)\}} (Q_1, Q_3, Q_4)$$

In the following, we note $\mathbb{Q}_\epsilon = (\epsilon, \ldots, \epsilon)$ the empty multi-prefix. In the case where we consider states in an automaton, $\mathbb{Q}_\epsilon$ would be any set of initial states. In our example, $\mathbb{Q}_\epsilon = (Q_0, Q_3, Q_4)$.

We then define a *multi-transition* $\delta = (\mathbb{Q}, E) \overset{e/F_{new}}{\rightarrow} (\mathbb{Q}', E')$ whenever there exist two multi-sets of events E_{new} and E^{org}_{new} such that we have:

$$\begin{cases} e \neq \perp \Rightarrow e \in E & (1) \\ \mathbb{Q} \overset{e/F_{new}/E_{new}}{\Rightarrow} \mathbb{Q}' & (2) \\ E^{org}_{new} = E_{new} \cap Events & (3) \\ E' = (E \backslash \{e\}) \cup \widetilde{E^{org}_{new}} \cup (E_{new} \backslash E^{org}_{new}) & (4) \end{cases}$$

In this transition, we ensure that (1) the triggering event is either the empty event or is in the buffer-pool E, (2) we advance an organ according to the triggering event, it produces a set of events E_{new}, (3) we select the events that are transferred between the organs, (4) the buffer-pool is updated: we remove[10] the triggering event, we add the input events corresponding to all the outputs between organs and finally we add the events that are outside the organs (in order to check them later with the system behavior).

Let us build such a multi-transition with our example. If we take $\mathbb{Q} = (Q_0, Q_3, Q_4)$ and $E = \{(msg, a), (msg, b)\}$, then we can build a transition with $e = (msg, a)$ in order to fulfill condition (1). We then have a transition fulfilling (2) with $F_{new} = \{f()\}$ and $E_{new} = \{(ack_A, a)\}$. As $(ack_A, a) \in Events$, we have $E^{org}_{new} = E_{new} = \{(ack_A, a)\}$ according to (3). Then $\widetilde{(ack_A, a)} = (ack_A, \alpha)$ and thus $\widetilde{E^{org}_{new}} = \{(ack_A, \alpha)\}$. As $(E \backslash \{e\}) = \{(msg, b)\}$ and $(E_{new} \backslash E^{org}_{new}) = \emptyset$, we have $E' = \{(ack_A, \alpha), (msg, b)\}$. Finally, we have the following multi-transition:

$$((Q_0, Q_3, Q_4), \{(msg, a), (msg, b)\}) \xrightarrow{(msg,a)/\{f()\}} ((Q_0, Q_3, Q_4), \{(ack_A, \alpha), (msg, b)\})$$

3.2.4 Multi-executions. A *multi-execution* is an execution composed with multi-transitions. We note $MExec(X_1, \ldots, X_n)$ the set of multi-executions constructed on the executions $X_1, \ldots, X_n$. Moreover, we note:

- $MExec_{(\mathbb{Q},E)\rightarrow}(X_1, \ldots, X_n)$ the set of multi-executions starting with $(\mathbb{Q}, E)$,
- $MExec_{\rightarrow(\mathbb{Q},E)}(X_1, \ldots, X_n)$ the set of multi-executions ending on $(\mathbb{Q}, E)$,
- $MExec_{(\mathbb{Q},E)\rightarrow(\mathbb{Q}',E')}(X_1, \ldots, X_n)$ the set of multi-executions starting with $(\mathbb{Q}, E)$ and ending on $(\mathbb{Q}', E')$.

[9] The same state can be in several executions with different predecessors and successors, without the ability to "jump" from one execution to the other.

[10] We remove one occurrence only since the pools are multi-sets.

This is a multi-execution of our example:

$$((Q_0, Q_3, Q_4), \{(msg, s)\}) \xrightarrow{(msg,s)/\emptyset}$$

$$((Q_1, Q_3, Q_4), \{(msg, a), (msg, b)\}) \xrightarrow{(msg,a)/\{f()\}}$$

$$((Q_1, Q_3, Q_4), \{(ack_A, \alpha), (msg, b)\}) \xrightarrow{(msg,b)/\{g()\}}$$

$$((Q_1, Q_3, Q_4), \{(ack_A, \alpha), (ack_B, \beta)\}) \xrightarrow{(ack_A,\alpha)/\emptyset}$$

$$((Q_2, Q_3, Q_4), \{(ack_B, \beta)\}) \xrightarrow{(ack_B,\beta)/\emptyset} ((Q_0, Q_3, Q_4), \{(ok, s)\})$$

We then have, in this example, a multi-execution u which, with $\vec{X} = (X_{IO}, X_A, X_B)$, belongs to:

$$MExec_{((Q_0,Q_3,Q_4),\{(msg,s)\})\rightarrow((Q_0,Q_3,Q_4),\{(ok,s)\})}\vec{X}$$

3.2.5 Refinement. Given a multi-set E, we note $E_{\downarrow in}$ the multi-set of elements of E which are system input events and $E_{\downarrow out}$ the multi-set of elements of E which are system output events.

With those objects, we define the notion of *refinement* between organic composition and system execution. We say that a *multi-execution* u is a refinement of a system transition $\delta = q \xrightarrow{e/F_{new}/E_{new}} q'$ along $((\mathbb{Q}, E), (\mathbb{Q}', E'))$ and we note $refine_{(\mathbb{Q},E)\rightarrow(\mathbb{Q}',E')}(\delta, u)$ if we have:

$$\begin{cases} e \neq \perp \Rightarrow \\ \quad u \in MExec_{(\mathbb{Q},E_{\downarrow in}\cup\{e\})\rightarrow(\mathbb{Q}',E')}(X_1, \ldots, X_n) \wedge e \notin E' \\ e = \perp \Rightarrow u \in MExec_{(\mathbb{Q},E_{\downarrow in})\rightarrow(\mathbb{Q}',E')}(X_1, \ldots, X_n) \\ E'_{\downarrow out} = E_{new} \end{cases}$$

Considering $\delta_{sys} = q \xrightarrow{(msg,s)/\{\phi\}/\{(ok,s)\}} q'$ and our multi-execution u, we can have the above requirements if we take $\mathbb{Q} = \mathbb{Q}' = (Q_0, Q_3, Q_4)$, $E = \emptyset$ and $E' = \{(ok, s)\}$, thus we have:

$$refine_{((Q_0,Q_3,Q_4),\emptyset)\rightarrow((Q_0,Q_3,Q_4),\{(ok,s)\})}(\delta_{sys}, u)$$

Given a system execution $x = \delta_1 \cdot \ldots \cdot \delta_n$, a sequence of multi-executions $(u_1, \ldots, u_n)$ refines x, noted $refine(x, (u_1, \ldots, u_n))$, if there exist multi-sets $E_0, \ldots, E_n$ and prefixes $\mathbb{Q}_0, \ldots, \mathbb{Q}_n$ belonging to $pref(X_1) \times \ldots \times pref(X_n)$ such that:

$$\begin{cases} E_0 = \emptyset \\ \mathbb{Q}_0 = \mathbb{Q}_\epsilon \\ \forall i > 0, refine_{(\mathbb{Q}_{i-1},E_{i-1})\rightarrow(\mathbb{Q}_i,E_i)}(\delta_i, u_i) \end{cases}$$

In our example, if x is the execution consisting in only one occurrence of δ_{sys}, then we have $refine(x, u)$. This is possible because $(Q_0, Q_3, Q_4) \in \mathbb{Q}_\epsilon$.

3.2.6 Requirements. With those structures, we can state the following requirements:

Any system execution has an organic equivalent
To check this, we must ensure that for any system execution x, there are some multi-executions $u_1, \ldots, u_n$ belonging to $MExec(X_1, \ldots, X_n)$ such that we have $refine(x, (u_1, \ldots, u_n))$.

Any emerging behavior has a system equivalent
Although some organic behavior can be invisible at the black box level (and in this case, we permit that it was not specified in the system behavior), we have to ensure that the composition of organic behaviors does not introduce a new way to react to outside stimuli. To do that, we check that whenever a multi-execution is extended, there already is a system execution corresponding to this extension. The formal description of this requirement is out of the scope of this paper.

This section presented only two generic soundness requirements. BDMs include additional soundness requirements related to other sub-models which are out of the scope of this paper due to lack of space. BDMs also allow to express more domain-specific completeness requirements through solicitations, such as, for embedded systems, requiring to explicit the behavior of the system depending on the energy sources available. In complement, BDMs related to embedded systems would often include a functional view (Sect. 2.4) focused on the events related to variations in the energy sources provided to the system.

4 RELATED WORKS

Models' coherence verification is not a new challenge. Indeed, there are many existing approaches to address this problem. In this section we only report the works that, like our own, are focused on the process of modeling and verification of the particular category of properties that are inherent to the models regardless of the modeled system.

A noticeable fact when studying techniques that address these challenges is that most of them focus on verifying UML/OCL models [28]. This is, however, no surprise since UML as well as OCL are de facto standards for systems' modeling. These approaches can be distinguished according to different points of interest, for instance, the properties covered. These approaches can cover satisfiability properties [3, 4, 8, 9, 12, 38, 41], constraints-related properties [22, 23] or both [11, 27, 46–48, 54].

These techniques can also be distinguished according to the extent to which they support the OCL language. Specifically, we can identify a first category of approaches that support the full OCL language [3, 4, 8, 9, 11, 24–27, 47, 48, 54, 56, 57], a second category that supports a subset of OCL [13, 46, 49] and a third one that provides no support for OCL [12, 22, 23, 38, 41]. We can also distinguish these approaches with regard to the underlying verification techniques. For instance, approaches can rely on CSP [11, 12, 27], Description logics [12, 46], Relational logics [3, 4, 24–26], HOL [2, 8, 9, 49], process algebras [42, 52], etc. Other criteria can be considered such as tool support, assistance in correction of detected problems, termination guarantee.

It is worth noting that other works deal with the verification of validity requirements on other types of models. For instance, Weckesser et al. address the problem of checking the consistency of Clafer models [59, 60], Bertram et al. check the consistency of Component and Connectors view models [6, 40] and Schrefl et al. verify the consistency of behavior diagrams [55]. An important aspect common to all the aforementioned approaches is that the proposed verification techniques are closely correlated to the modeling language (UML/OCL, Claffer, C&C views and behavior diagrams). On the contrary, the work described in this paper is independent of

any particular modeling language thanks to the agnostic approach used to introduce the semantic domain of BDMs. Still, this work is applicable to a variety of modeling formalisms by mapping the conceptual elements of these formalisms to BDM.

Another important difference with these techniques is that they all (except for [55]) focus on static models while our interest also includes the behavioral aspects of the system. This particular concern has been addressed in another family of works. Specifically, most related to our own description of behavioral decomposition are the works on Interface and Input/Output Automata [16, 39]. Interface Automata provide formal technical tools to express and check the compatibility of the components of a given system and to verify that their combination actually produces the overall desired behavior. This is achieved by providing a coherent theory consisting in a semantic domain, a refinement operator and an equivalence relation. The base formalism has been extended to address various issues such as error propagation in system refinements [21], compositional verification [64] or requirements traceability [18]. Although we partially share with this family of techniques some of the objectives, the behavioral decomposition sub models, the particular semantics of synchronization mechanism adopted in the case of our semantic domain and the distinction between *must* and *may* transitions in interface theories make their exploitation in our approach somehow cumbersome.

Finally, although only outlined in this article, functional views are an important part of the work on BDMs. It is worth noting that we understand views in our current work as slices of the system. In this sense, our work is comparable to previous ones [1, 10, 34, 45, 50]. However, a notable difference with these approaches is that our interest is not slices generation, slices update, nor generation of slices that are syntactically valid with respect to the original metamodels. Our requirements concern the semantic conformance of the functional views/slices with the original global model of the system.

5 CONCLUSION AND FUTURE WORKS

In this paper, we proposed a mathematical framework to design agnostic models called Behavior Decomposition Model. We showed how the submodels of a BDM can be used to describe the black-box behavior of a system and its first-level decomposition. Our approach provides two algorithms to check some internal coherence properties. Firstly, the adherence of the black-box behavior to a series of solicitations enabling to formally draw the perimeter of this behavior. Secondly, the coherence between the black-box level and the first-level of description, both in terms of behavior and function calls.

In present article, we essentially focused on two important sub models of BDMs. However, another important aspect, related to the decomposition of the black-box functions into first-level functions still needs some refinement. Moreover, work is in progress to provide automatic translation tools between our formalism and general purpose modeling tools like SysML.

ACKNOWLEDGMENTS

This work was in part conducted at Capgemini Rennes, France, and benefited from discussions with other members of the team including Cédric Lahellec, Gidlas Premel-Cabic, Evelyne Kabore and team leader Hélène François. The authors are also grateful to the reviewers for their helpful comments.

REFERENCES

[1] Reza Ahmadi, Ernesto Posse, and Juergen Dingel. 2018. Slicing UML-based Models of Real-time Embedded Systems. In *Int. Conf. Model Driven Engineering Languages and Systems*. ACM, 346–356.
[2] Tamleek Ali, Mohammad Nauman, and Masoom Alam. 2007. An Accessible Formal Specification of the UML and OCL Meta-Model in Isabelle/HOL. In *Int. Multitopic Conf.* IEEE. https://doi.org/10.1109/inmic.2007.4557693
[3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2007. UML2Alloy: A challenging model transformation. In *Int. Conf. Model Driven Engineering Languages and Systems*. Springer, 436–450.
[4] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2008. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling* 9, 1 (dec 2008), 69–86. https://doi.org/10.1007/s10270-008-0110-3
[5] Alessandro Artale, Diego Calvanese, and Angélica Ibánez-Garcıa. 2010. Checking full satisfiability of conceptual models. In *Int. Work. Description Logics*. 55.
[6] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. 2017. Component and Connector Views in Practice: An Experience Report. In *Int. Conf. Model Driven Engineering Languages and Systems*. IEEE, 167–177.
[7] Gregor v. Bochmann, Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. 2008. Testing Systems Specified as Partial Order Input/Output Automata. In *Testing of Software and Communicating Systems*, Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–183.
[8] Achim D Brucker, Jürgen Doser, and Burkhart Wolff. 2007. An MDA framework supporting OCL. *Electronic Communications of the EASST* 5 (2007).
[9] Achim D. Brucker and Burkhart Wolff. 2008. HOL-OCL: A Formal Proof Environment for uml/ocl. In *Fundamental Approaches to Software Engineering*, José Luiz Fiadeiro and Paola Inverardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–100.
[10] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A Feature-based Survey of Model View Approaches. *Software and Systems Modeling* 18, 3 (June 2019), 1931–1952. https://doi.org/10.1007/s10270-017-0622-9
[11] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Int. Conf. Automated Software Engineering*. ACM, 547–548.
[12] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. 2007. Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. In *Artificial Intelligence and Human-Oriented Computing*, Roberto Basili and Maria Teresa Pazienza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–47.
[13] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. 2010. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST* 24 (2010).
[14] Carolina Dania and Manuel Clavel. 2013. OCL2FOL+: Coping with Undefinedness. *OCL@ MoDELS* 1092 (2013), 53–62.
[15] Carolina Dania and Manuel Clavel. 2016. OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Int. Conf. Model Driven Engineering Languages and Systems*. ACM, 65–75.
[16] Luca De Alfaro and Thomas A Henzinger. 2001. Interface automata. In *SIGSOFT Software Engineering Notes*, Vol. 26. ACM, 109–120.
[17] Rocco De Nicola and Roberto Segala. 1995. A process algebraic view of input/output automata. *Theoretical Computer Science* 138, 2 (1995), 391 – 423. https://doi.org/10.1016/0304-3975(95)92307-J Meeting on the mathematical foundation of programing semantics.
[18] Iulia Dragomir, Iulian Ober, and Christian Percebois. 2017. Contract-based modeling and verification of timed safety requirements within SysML. *Software & Systems Modeling* 16, 2 (2017), 587–624.
[19] Elfriede Dustin, Jeff Rashka, and John Paul. 1999. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[20] Peter Feiler, David Gluch, and John Hudak. 2006. *The Architecture Analysis & Design Language (AADL): An Introduction*. Technical Note CMU/SEI-2006-TN-011. Carnegie Mellon University.
[21] Sascha Fendrich and Gerald Lüttgen. 2016. A generalised theory of interface automata, component compatibility and error. In *Int. Conf. Integrated Formal Methods*. Springer, 160–175.
[22] Anna Formica. 2002. Finite satisfiability of integrity constraints in object-oriented database schemas. *Transactions on Knowledge and Data Engineering* 14, 1 (2002), 123–139.
[23] Anna Formica. 2003. Satisfiability of object-oriented database constraints with set and bag attributes. *Information Systems* 28, 3 (may 2003), 213–224. https:

//doi.org/10.1016/s0306-4379(02)00010-8
[24] Martin Gogolla, Jørn Bohling, and Mark Richters. 2005. Validating UML and OCL models in USE by automatic snapshot generation. *Software & Systems Modeling* 4, 4 (01 Nov 2005), 386–398. https://doi.org/10.1007/s10270-005-0089-y
[25] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1-3 (2007), 27–34.
[26] Martin Gogolla, Lars Hamann, and Mirco Kuhlmann. 2010. Proving and Visualizing OCL Invariant Independence by Automatically Generated Test Cases. In *Tests and Proofs*, Gordon Fraser and Angelo Gargantini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–54.
[27] Carlos A González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Int. Work. Formal Methods in Software Engineering: Rigorous and Agile Approaches*. IEEE, 44–50.
[28] Carlos A González and Jordi Cabot. 2014. Formal verification of static software models in MDE: A systematic review. *Information and Software Technology* 56, 8 (2014), 821–838.
[29] ISO/PAS. 2015. *Automation systems and integration — Object-Process Methodology*. Standard 19450:2015. International Organization for Standardization. https://www.iso.org/standard/62274.html
[30] ITU. 1998. *Formal semantics of message sequence charts*. Recommendation Z.120 Annex B. International Telecommunication Union. https://www.itu.int/rec/T-REC-Z.120-199804-I!AnnB/en
[31] ITU. 2011. *Message Sequence Chart (MSC)*. Recommendation Z.120. International Telecommunication Union. https://www.itu.int/rec/T-REC-Z.120-201102-I/en
[32] ITU. 2019. *Specification and Description Language - Overview of SDL-2010*. Recommendation Z.100. International Telecommunication Union. https://www.itu.int/rec/T-REC-Z.100-201910-I/en
[33] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2010. *The Theory of Timed I/O Automata* (2 ed.). Morgan & Claypool. https://doi.org/10.2200/S00310ED1V01Y201011DCT005
[34] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. 2016. Automatically deriving the specification of model editing operations from meta-models. In *Int. Conf. Theory and Practice of Model Transformations*. Springer, 173–188.
[35] Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to relational logic and back. In *Int. Conf. Model Driven Engineering Languages and Systems*. Springer, 415–431.
[36] Kim G Larsen, Ulrik Nyman, and Andrzej Wasowski. 2007. Modal I/O Automata for Interface and Product Line Theories. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 64–79.
[37] Gurvan Le Guernic. 2016. Modeling requirements should be language agnostic! Example of a formal definition of simple Behavioral Decomposition Models. In *Int. Conf. Model-Driven Engineering and Software Development*. IEEE, 555–562.
[38] Maurizio Lenzerini and Paolo Nobili. 1990. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems* 15, 4 (jan 1990), 453–461. https://doi.org/10.1016/0306-4379(90)90048-t
[39] Nancy A Lynch and Mark R Tuttle. 1989. An Introduction to Input/Output Automata. *CWI Quaterly* 2, 3 (Sept. 1989), 219–246.
[40] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2014. Verifying component and connector models against crosscutting structural views. In *Int. Conf. Software Engineering*. ACM, 95–105.
[41] Azzam Maraee and Mira Balaban. 2007. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *Eur. Conf. Model Driven Architecture-Foundations and Applications*. Springer, 17–31.
[42] R Marcano and N Levy. 2002. Using B formal specifications for analysis and verification of UML/OCL models. In *Work. consistency problems in UML-based software development*. Citeseer, 91–105.
[43] OMG. 2017. *OMG® Unified Modeling Language® (OMG UML®)*. Standard. Object Management Group (OMG). http://www.omg.org/spec/UML/2.5.1/ Version 2.5.1.
[44] OMG. 2019. *OMG Systems Modeling Language (OMG SysML^{TM})*. Standard. Object Management Group (OMG). http://www.omg.org/spec/SysML/1.6/ Version 1.6.
[45] Christopher Pietsch, Manuel Ohrndorf, Udo Kelter, and Timo Kehrer. 2017. Incrementally slicing editable submodels. In *Int. Conf. Automated Software Engineering*. IEEE Press, 913–918.
[46] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. 2012. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering* 73 (mar 2012), 1–22. https://doi.org/10.1016/j.datak.2011.09.004
[47] Anna Queralt, Guillem Rull, Ernest Teniente, Carles Farré, and Toni Urpí. 2010. AuRUS: Automated Reasoning on UML/OCL Schemas. In *Conceptual Modeling*, Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 438–444.
[48] Anna Queralt and Ernest Teniente. 2006. Reasoning on UML Class Diagrams with OCL Constraints. In *Conceptual Modeling*, David W. Embley, Antoni Olivé, and Sudha Ram (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 497–512.
[49] Lukman Ab. Rahim. 2008. Mapping from OCL/UML metamodel to PVS metamodel. In *Int. Symp. Information Technology*. IEEE. https://doi.org/10.1109/itsim.2008.4631599
[50] Michaela Rindt, Timo Kehrer, and Udo Kelter. 2014. Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. *Demos@MoDELS* (2014).
[51] David Roe, Krysia Broda, and Alessandra Russo. 2003. *Mapping UML models incorporating OCL constraints into Object-Z*. Imperial College of Science, Technology and Medicine, Department of Computing.
[52] David Roe, Krysia Broda, and Alessandra Russo. 2003. *Mapping UML models incorporating OCL constraints into Object-Z*. Imperial College of Science, Technology and Medicine, Department of Computing.
[53] Pascal Roques. 2016. MBSE with the ARCADIA Method and the Capella Tool. In *Eur. Cong. Embedded Real Time Software and Systems*. https://web1.see.asso.fr/erts2016/uploads/program/paper_5.pdf
[54] Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí. 2008. Providing explanations for database schema validation. In *Int. Conf. Database and Expert Systems Applications*. Springer, 660–667.
[55] Michael Schrefl and Markus Stumptner. 2002. Behavior-consistent specialization of object life cycles. *Transactions on Software Engineering and Methodology* 11, 1 (2002), 92–148.
[56] Mathias Soeken, Robert Wille, and Rolf Drechsler. 2011. Encoding OCL data types for SAT-based verification of UML/OCL models. In *Int. Conf. Tests and Proofs*. Springer, 152–170.
[57] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. 2010. Verifying UML/OCL models using Boolean satisfiability. In *Conf. Design, Automation and Test in Europe*. European Design and Automation Association, 1341–1344.
[58] Marcin Szlenk. 2006. Formal semantics and reasoning about uml class diagram. In *Int. Conf. Dependability of Computer Systems*. IEEE, 51–59.
[59] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. 2017. Towards complete consistency checks of clafer models. In *Int. Work. Feature-Oriented Software Development*. ACM, 11–20.
[60] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. 2018. Mathematical Programming for Anomaly Analysis of Clafer Models. In *Int. Conf. Model Driven Engineering Languages and Systems*. ACM, 34–44.
[61] Robert Wille, Mathias Soeken, and Rolf Drechsler. 2012. Debugging of inconsistent UML/OCL models. In *Conf. Design, Automation and Test in Europe*. EDA Consortium, 1078–1083.
[62] Hao Wu. 2017. Finding achievable features and constraint conflicts for inconsistent metamodels. In *Eur. Conf. Modelling Foundations and Applications*. Springer, 179–196.
[63] Hao Wu. 2017. MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *Int. Conf. Integrated Formal Methods*. Springer, 348–356.
[64] Fuyuan Zhang, Yongwang Zhao, David Sanán, Yang Liu, Alwen Tiu, Shang-Wei Lin, and Jun Sun. 2018. Compositional Reasoning for Shared-Variable Concurrent Programs. In *Int. Symp. Formal Methods*. Springer, 523–541.